DOI: 10.15514/ISPRAS-2026-38(3)-21

DPN Verifier: A Toolkit for Faster Soundness Verification and Repair of Process Models with Data

N.M. Suvorov, ORCID: 0000-0003-2871-9757 <nmsuvorov@hse.ru>
HSE University,
20, Myasnitskaya st., Moscow, 101000, Russia.

Abstract. Data Petri Nets (DPNs) extend classical Petri nets to model processes where data directly influences control-flow, enabling a comprehensive view of system behavior and possibility to detect failure points that could otherwise be hidden. Soundness is a correctness criterion that captures such failure points as deadlocks and livelocks as well as model boundedness and absence of dead activities. This paper introduces a novel soundness verification technique for DPNs that achieves significant efficiency by requiring only two state space constructions. This advancement makes soundness verification and repair applicable to larger, more complex models. We have implemented verification and repair algorithms using this technique in the DPN Verifier toolkit, a versatile tool that supports multi-perspective model analysis through various state-space structures and abstraction levels. For practicality and interoperability, the toolkit imports and exports DPNs and state-spaces using dedicated formats proposed in this paper and is delivered as a desktop application, a console application, and a class library, moving beyond a mere research prototype to offer a multifaceted platform ready for both academic and industrial use. The results of performance evaluation demonstrate that our implementation achieves lower verification and repair times for most DPNs from the literature compared to existing solutions, confirming its practical value for realistic applications.

Keywords: process models; data-aware processes; data Petri nets; soundness verification; soundness repair.

For citation: Suvorov N.M. DPN Verifier: A Toolkit for Faster Soundness Verification and Repair of Process Models with Data. Trudy ISP RAN/Proc. ISP RAS, vol. 38, issue 3, part 2, 2026, pp. 49-66. DOI: 10.15514/ISPRAS-2026-38(3)-21.

Acknowledgements. This study has been supported by the Basic Research Program at HSE University, Russia.

DPN Verifier: Инструментарий для ускоренной верификации и исправления дефектных моделей процессов с данными

Н.М. Суворов, ORCID: 0000-0003-2871-9757 <nmsuvorov@hse.ru>
Национальный исследовательский университет «Высшая школа экономики»,
Россия, 101000, г. Москва, ул. Мясницкая, д. 20.

Аннотация. Сети Петри с данными (DPN) являются расширением классических сетей Петри, позволяющим моделировать процессы, где данные влияют на поток управления, обеспечивая комплексное представление о поведении системы и возможность обнаружения точек отказа, которые в противном случае были бы скрыты. Одним из критериев корректности для моделей процессов является бездефектность. Модель процесса называется бездефектной, если она всегда корректно завершается и каждое действие модели представлено хотя бы в одном исполнении процесса. В данной статье представлен новый метод проверки бездефектности DPN, который требует не более двух построений пространства состояний, что существенно ниже по сравнению с предложенными и реализованными ранее алгоритмами. Это усовершенствование делает проверку бездефектности и исправление дефектных моделей применимым к моделям достаточно больших размеров. Мы реализовали алгоритмы верификации и исправления, использующие этот метод, в DPN Verifier – универсальном инструменте, поддерживающем анализ моделей посредством различных структур пространств состояний и уровней абстракции. Инструментарий позволяет импортировать и экспортировать как DPN, так и структуры пространств состояний с использованием специальных форматов, предложенных в данной статье, и поставляется в виде настольного приложения, консольного приложения и библиотеки классов, что делает инструмент применимым как для академического, так и для промышленного использования. Результаты экспериментов демонстрируют более высокую скорость нашей реализации алгоритмов верификации и исправления для большинства DPN, описанных в литературе, по сравнению с существующими решениями, что подтверждает практическую ценность предложенного нами решения для реальных приложений.

Ключевые слова: модели процессов; процессы, обогащенные данными; сети Петри с данными; верификация бездефектности; исправление дефектных моделей.

Для цитирования: Суворов Н. М. DPN Verifier: Инструментарий для ускоренной верификации и исправления дефектных моделей процессов с данными. Труды ИСП РАН, том 38, вып. 3, часть 2, 2026 г., стр. 49–66 (на английском языке). DOI: 10.15514/ISPRAS–2026–38(3)–21.

Благодарности. Выполнение исследований поддержано Программой фундаментальных исследований Национального исследовательского университета «Высшая Школа Экономики», Россия.

1. Introduction

Distributed processes often rely on data. The data can be manipulated by process activities and referenced at various decision points. This critical aspect is addressed by several data-aware modeling formalisms, such as workflow nets with data (WFD-nets) [1], workflow nets with tables (WFT-nets) [2], and Data Petri nets (DPNs) [3], and frameworks, the most well-known of which is BPMN 2.0 [4]. The usage of Petri net extensions is common for formal process model analysis due to the simplicity and unambiguity of these formalisms. In this work, we focus on models represented as DPNs, a Petri net extension that represents the interplay of data and control flows by transition constraints that define input and output conditions on global variables.

Despite the approach used to model a process, a constructed model may contain errors. Since some errors are domain-specific, it is important to verify a model at least for common domain-agnostic flaws, namely for deadlocks, livelocks, the presence of activities that are never executed, and unbounded resource growth or consumption. Soundness is a correctness property that captures all these types of errors. Specifically, a model is called sound if it always properly terminates and each process model activity occurs in at least one process instance [5]. Soundness verification is an important step before the process implementation that helps to detect failure points in a process.

Elimination of the errors detected at the verification stage is called soundness repair. Conducting soundness verification and repair right after the model design can help to avoid potential problems in the process to be implemented. This also works for models represented as DPNs. Consider Fig. 1, which represents a visit to a casino. The model has a deadlock when there is a token in p_2 and age is not more than 18: a visitor registers but does not receive a pass. It is reasonable to omit such a deadlock in a process. A straightforward repair approach here is to restrict the constraint of $Register$ and forbid its execution if $age \leq 18$.

A significant body of research has been dedicated to soundness verification and repair of DPNs [6][12]. At the current state, there exist two research prototypes, namely [10] and [12], that implement verification and repair algorithms. However, both the prototypes have several limitations. First, the inner structure of used verification algorithms requires multiple state space constructions for checking model soundness, which narrows the application scope of verification and repair methods to models of moderate sizes. Second, each of the tools was developed as a proof-of-concept. Thus, as an example, if a model has a large state space and the visualization library cannot properly handle it, the user would never receive a result regarding model soundness. There are also other problems related to the proof-of-concept versions, including the impossibility to export verification/repair results and the necessity to use the proposed desktop/web interfaces, which makes it difficult to include the verification and repair techniques into model designing pipelines.

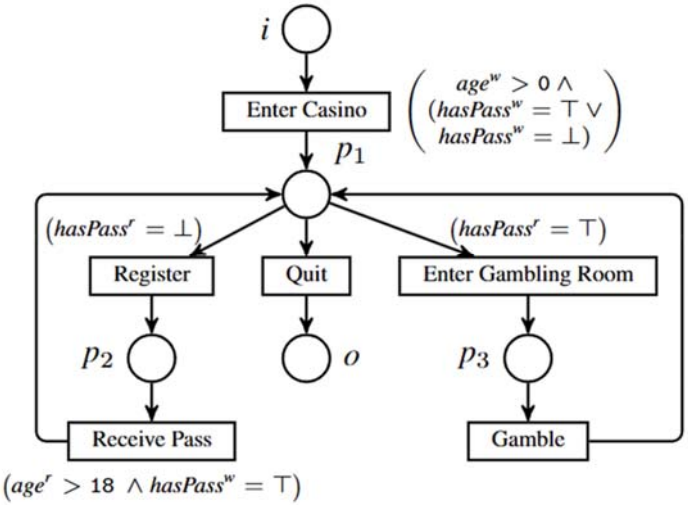

Fig. 1. A DPN representing a visit to a casino. The initial marking is [*i*]. *The final marking is* [*o*]. *Variable hasPass is of a boolean type, the age is of a real type.*

In this paper, we introduce a soundness verification toolkit, DPN Verifier. The toolkit is delivered in three forms: as a desktop application, as a console application, and as a class library, and is available for download on https://github.com/ConferenceParticipant/DPNVerifier. Compared to the previous implementations, it provides more extensive interoperability means, includes wider opportunities for model analysis, and allows for greater interaction opportunities, for instance, making the visualization optional when providing a soundness verification result. Last but not least, the tool implements a new verification algorithm, proposed in this paper, that does not construct the state space more than twice, which extends the scope of its use to process models of larger sizes.

The main research contributions of this work are as follows:

- A soundness verification and repair algorithms for DPNs based on a new refinement procedure that requires a single state space construction.

- The definitions of the PNMLX format used for the DPN import/export and of the ASML format used for the state space import/export.
- The DPN Verifier tool delivered in three different forms allowing the analysis of a DPN at different abstraction levels, the verification of different versions of soundness (classical and relaxed lazy [13]), and the repair of DPN soundness.
- Experimental evaluation that justifies the practical applicability of the DPN Verifier and the shorter verification and repair execution times compared to the existing solutions.

2. Related Work

Manually crafted process models are often prone to errors. Different papers, such as [14]-[16], investigate the sources of such errors and the reasons why they are made. Studies analyzing industrial and reference models have found significant error rates, ranging from 5.6% in the SAP reference model [17] to over 72% in practical industrial settings [16]. A substantial portion of errors can be detected and corrected through formal soundness verification and repair procedures.

For subtle and accurate analysis of processes, where data influences the process execution, the process model formalism should be able to capture the interplay of control and data flows. One of such formalisms is a DPN. Significant research has been conducted to make DPNs viable in practice, including techniques for model discovery from event logs [3], conformance checking [18][19], and soundness verification [6]-[9]. However, the proposed soundness verification algorithms are either applicable to models of moderate sizes, as algorithms [7][9], or to models with rather simple conditions, as algorithm [6]. We consider the setting when variables may be compared to constants or other variables; thus, we are interested in algorithms [7][9]. The verification technique from [7] performs a single state space construction, but constructs it as granular as possible, considering all possible combinations of markings and formulas. Algorithms [8][9] require multiple constructions of the state space; for instance, algorithm [8] constructs a state space for each reachable DPN marking. All the existing repair algorithms, namely [10][12], use one of the mentioned verification algorithms for verifying a model at each repair iteration, which entails a rather narrow scope of use for these algorithms due to the high execution times of the corresponding verification procedures.

Apart from the limitations of the verification and repair algorithms, a significant tooling gap also persists. Existing implementations of DPN algorithms are fragmented across different platforms, each with limitations. The process mining [3] and conformance checking [18] algorithms are implemented as plugins for ProM [20], an open-source framework for process mining algorithms. Although ProM is a powerful tool for process mining, it is widely often criticized for a confusing and outdated user interface, and therefore its use is gradually declining. Modern process mining algorithms are now mainly implemented as plugins for PM4Py [21], a Python-based library, which by its nature integrates well with the broader ecosystem of machine learning and data science tools. Interestingly, none of the existing algorithms for DPNs are implemented in PM4Py. Some authors strive to use Python but develop their own separate applications independent of the mentioned above library: CoCoMoT [19] for conformance checking and the Ada tool [12] for model verification and repair are examples of such implementations. The latter was one of the two implemented tools for model verification and repair, but maintenance has recently been discontinued. Another tool for verifying and repairing a model was implemented as a desktop application in C# [10], but as a research prototype, which is why its functionality and interoperability are very limited: for example, it is impossible to export the repaired model. Both the implementations used the verification algorithms that require multiple state space constructions.

Considering the limitations of the existing tools for soundness verification and repair, we decided to devise a new soundness verification method that requires not more than two constructions of the state space. The proposed method forms a basis in a new toolkit for soundness verification and repair of DPNs, introduced in this paper.

3. Data Petri Nets

A Data Petri net is a place/transition net that includes transition constraints that define input and output conditions over the data variables.

Each constraint φ over a set X of variables is an expression of the form

$$\varphi := \top \mid x \odot y \mid x \odot c \mid \neg\varphi' \mid \varphi_1 \wedge \varphi_2,$$

where: (i) $\top$ is the logical "true"; (ii) $x, y \in X$; (iii) $c \in \mathbb{R}$; (iv) $\odot \in \{<, =, >\}$, (v) $\varphi', \varphi_1, \varphi_2$ are constraints.

We make use of the following standard equivalences: (i) $\neg\top = \bot$; (ii) $\varphi_1 \vee \varphi_2 = \neg(\neg\varphi 1 \wedge \neg\varphi 2)$; (iii) $x \leq y = \neg(x > y)$; (iv) $x \geq y = \neg(x < y)$; and (v) $x \neq y = \neg(x = y)$. By $\Phi(X)$, we denote the language of constraints. For example, for $X = \{y, z\}$, expressions $z > y$, $y < 0$ and $(y > 2) \vee ((z < 4) \wedge (z = 1))$ are in $\Phi(X)$. The satisfaction of a constraint is defined in a standard way. We refer a reader for details to [9]. In this paper, we consider real-typed variables and constants. For representing transition guards, we use language $\Phi(V^r \cup V^w)$, where V^r and V^w represent read and written variables, respectively.

Definition 1 (Data Petri net) [6]. A *data Petri net* (DPN) is a tuple $\mathcal{N} = \langle P, T, F, V, guard \rangle$, where:

- P and T are disjoint sets of places and transitions, respectively;
- $F: (P \times T) \cup (T \times P) \to N$ is a flow relation;
- V is a finite set of variables;
- $guard: T \to \Phi(V^r \cup V^w)$ is a guard assignment function that labels transitions with constraints.

Given $t \in T$, we also define $read(t)$ and $write(t)$ to denote, respectively, all the variables from V^r and V^w that occur in $guard(t)$.

A state of a DPN $\mathcal{N}$ is a pair (M, α), where $M: P \to N$ is a marking function that assigns a number of tokens to each place $p \in P_{\mathcal{N}}$, and $\alpha: V \to \mathbb{R}$ is a variable valuation function that assigns a value to each variable in V. A DPN moves between states by firing (enabled) transitions. After a transition fires, a new state is reached, with a new corresponding marking and valuation.

Given a DPN $\mathcal{N}$ and a state (M, α), we say that transition $t \in T$ may fire at (M, α) yielding a new state (M', α'), denoted as $(M, \alpha)[t\rangle(M', \alpha')$, if and only if:

- $M(p) \geq F(p, t)$ and $M'(p) = M(p) - F(p, t) + F(t, p)$, for all $p \in P$;
- $\beta \vDash guard(t)$, where $\beta: V^r \cup V^w \to \mathbb{R}$ and, for every $v \in V$, it holds that $\beta(v^r) = \alpha(v)$ and $\beta(v^w) = \alpha'(v)$;
- $\alpha(v) = \alpha'(v)$, for every $v \in V$ such that $v^w \neq write(t)$.

This is naturally extended to finite sequences of transition firings $\sigma = t_1 \cdots t_n$, called traces, while each trace induces a run denoted as $(M_0, \alpha_0)[t_1\rangle \ldots [t_n\rangle(M_n, \alpha_n)$ (or, equivalently, as $(M_0, \alpha_0)[\sigma\rangle(M_n, \alpha_n)$). Given two states (M, α) and (M', α'), we write $(M, \alpha)[*\rangle(M', \alpha')$ to denote zero or more transition firings leading from (M, α) to (M', α'). We fix one state (M_I, α_I) as initial. By M_F, we denote the final marking. We call state (M, α) final if $M = M_F$. In what follows, we use $\mathcal{M}_{\mathcal{N}}$ to denote all the markings in the DPN $\mathcal{N}$. Given two markings M' and M'' of a DPN $\mathcal{N}$, we write $M'' \geq M'$ if and only if for all $p \in P_{\mathcal{N}}$, we have $M''(p) \geq M'(p)$, and we write $M'' \succ M'$ if and only if $M'' \geqslant M'$ and there exists $p \in P_{\mathcal{N}}$ s.t. $M''(p) > M'(p)$.

Definition 2 (Reachability set). Let $\mathcal{N}$ be a DPN with an initial state (M_I, α_I). The reachability set of $\mathcal{N}$, denoted by $Reach_{\mathcal{N}}$, is the smallest set of states, which is inductively defined as follows:

- $(M_I, \alpha_I) \in Reach_{\mathcal{N}}$;
- if $(M, \alpha)[t\rangle(M', \alpha')$ for $t \in T$ and $(M, \alpha) \in Reach_{\mathcal{N}}$, then $(M', \alpha') \in Reach_{\mathcal{N}}$.

In the following, we will be interested in the boundedness property of DPNs. We say that a DPN $\mathcal{N}$ is bounded if there exists a bound $k \in N$ such that $M(p) \leq k$, for all $p \in P$ and $(M, \alpha) \in Reach_{\mathcal{N}}$.

Using the reachability set, we can provide a definition to soundness:

Definition 3 (Soundness) [6]. Let $\mathcal{N}$ be a DPN with initial state (M_I, α_I) and final marking M_F. $\mathcal{N}$ is sound if and only if the following properties hold:

- for each $(M, \alpha) \in Reach_{\mathcal{N}}$, there exists α' s.t. $(M, \alpha)[*\rangle(M_F, \alpha')$.
- for each $(M, \alpha) \in Reach_{\mathcal{N}}$, $M \geq M_F \Rightarrow M = M_F$.
- for each $t \in T$, there exist $(M_1, \alpha_1), (M_2, \alpha_2) \in Reach_{\mathcal{N}}: (M_1, \alpha_1)[t\rangle(M_2, \alpha_2)$.

As an example, the DPN from Fig. 1 is unsound, since the property that ensures the reachability of the final state is violated. One of the executions that lead to a deadlock is the following: $([i], \{age = 0, hasPass = \bot\})$ $[Enter\ Casino\rangle$ $([p_1], \{age = 14, hasPass = \bot\})[Register\rangle$ $([p_2], \{age = 14, hasPass = \bot\})$.

The number of states in $Reach_{\mathcal{N}}$ can be infinite even if the DPN is bounded, especially for models with real-typed variables. Thus, a classical reachability graph can barely be used for soundness verification in practice. In the following sections, we introduce the abstract state space structures that can be used for analyzing the behavioral properties of DPNs, including model soundness, and that could be constructed in the DPN Verifier.

3.1 PNMLX format

The classical PNML format [22] does not allow for defining variables or transition constraints. Thus, we extended this format to make it sufficient for DPN representation. First, the net now includes a set of variables with *variableType* defined in listing 1.

```
<xs:complexType name="variableType">
    <xs:sequence>
        <xs:element name="name" type="xs:string"/>
    </xs:sequence>
    <xs:attribute name="type" use="required">
        <xs:simpleType>
            <xs:restriction base="xs:string">
                <xs:enumeration value="Real"/>
                <xs:enumeration value="Integer"/>
                <xs:enumeration value="Boolean"/>
            </xs:restriction>
        </xs:simpleType>
    </xs:attribute>
</xs:complexType>
```

Listing 1. Type 'variableType' definition.

Second, a transition now includes an optional string attribute `guard` that defines a transition constraint. The constraint is an expression that uses logical connectives && (and) and || (or) to combine atomic variable-operator-constant and/or variable-operator-variable conditions. Read variables have a subscript `_r`, write variables have a subscript `_w`. String `"hasPass_w == True && age_r > 18"` is an example of an admissible value for the guard. A simplified Backus-Naur Form for admissible expressions is shown in Listing 2. The described format is used both for DPN import and export.

4. Means for Data Petri Net Analysis

Analysis of DPN behavioral properties requires the state space structures that are finite despite the finiteness of the domains of variables. In what follows, we define some of them, show how they are implemented in the DPN Verifier, and describe how they can be used for soundness verification.

```
<expression>       ::= <or_expression>
<or_expression>    ::= <and_expression> ( "||" <and_expression> )*
<and_expression>   ::= <atomic_expr> ( "&&" <atomic_expr> )*
<atomic_expr>      ::= <comparison> | ("!")? "(" <expression> ")"
<comparison>       ::= <real_var> <num_op> <real_val>       |
                       <int_var>  <num_op>  <int_val>       |
                       <bool_var> <bool_op> <bool_val>
<num_op>           ::= ">" | ">=" | "<" | "<=" | "==" | "!="
<bool_op>          ::= "==" | "!="
<real_val>         ::= <real_var> | <real_constant>
<int_val>          ::= <int_var> | <int_constant>
<bool_val>         ::= <bool_var> | <bool_constant>
<real_var>         ::= <name> "_" <access_suffix>
<int_var>          ::= <name> "_" <access_suffix>
<bool_var>         ::= <name> "_" <access_suffix>
<name>             ::= [a-zA-Z_][a-zA-Z0-9_]*
<access_suffix>    ::= "r" | "w"
<real_constant>    ::= ("-")? [0-9]+ ("." [0-9]+)?
<int_constant>     ::= ("-")? [0-9]+
<bool_constant>    ::= "True" | "False"
```

Listing 2. Admissible expressions definition.

4.1 Classical State Space Abstractions

A straightforward way to tame the infiniteness of variable valuations is to use a generalization of a state space, where each node would represent not a single state, but a set of states, which have the same marking but different variable valuations. Since sets of variable valuations may still be infinite, we can use formulas of $\Phi(V)$ to describe them (if language Φ is expressive enough [9]).

First, we need a procedure to compute a formula for the next state in a state space generalization. Let $\phi \in \Phi(V)$ be a formula of the current state, t be some transition, and V_o be a set of variables that are simultaneously read and written by t. To compute the formula of the state resulting from executing t on the state with ϕ, we need to eliminate the existential quantifier from $\exists V_o(\phi[v/v^r] \wedge guard(t))$ and substitute v^r, v^w in the resulting formula for v. We denote this operation as $\phi \oplus guard(t)$. In our tool, results of quantifier eliminations are computed using the SMT-solver Z3 [23]. Using this operation, we can now define an abstract reachability graph (ARG):

Definition 4 (Abstract Reachability Graph). Let $\mathcal{N} = \langle P, T, F, V, guard \rangle$ be a DPN with initial state (M_I, α_I). Let $\Phi(V)$ be the language of constraints, as in Section 3. Abstract Reachability Graph $ARG_{\mathcal{N}}$ of $\mathcal{N}$ is a tuple $\langle S, E, s_0 \rangle$, where:

- $s_0 = (M_I, \phi_I) \in S$ is the initial node with $\phi_I = \bigwedge_{v \in V}\{v = \alpha_I(v)\}$;
- $S \subseteq \mathcal{M}_{\mathcal{N}} \times \Phi(V)$ is the least set that contains s_0 and is closed under the transition relation;
- $E \subseteq S \times T \times S$ is a set of arcs labeled with transitions, s.t. $((M, \phi), t, (M', \phi')) \in E$ if and only if (i) $\phi' = \phi \oplus guard(t)$ and $[[\phi]] \neq \emptyset$, (ii) for each $p \in P$, $M(p) \geq F(p,t)$, (iii) $M'(p) = M(p) - F(p,t) + F(t,p)$.

The ARG is infinite if a DPN is unbounded. If a DPN is unbounded, our tool terminates when a strictly covering node is found and a fragment of the DPN is returned to a user. The covering and strict covering relations are defined as follows.

Definition 5 (Coverability). Let (M, φ), (M', φ') be two nodes. We say that (M', φ') covers (resp., strictly covers) (M, φ), denoted as $(M, \varphi) \sqsubseteq (M', \varphi')$ (resp., $(M, \varphi) \sqsubset (M', \varphi')$) if and only if the sets of all assignments that satisfy φ and φ' are equal and $M \preceq M'$ (resp., $M \prec M'$).

The algorithm 1 implemented in the DPN Verifier toolkit for constructing an ARG is defined below.

Input: A DPN $\mathcal{N} = \langle P, T, F, V, guard \rangle$ with initial state (M_I, α_I).
Result: $(ARG_{\mathcal{N}}, isFull)$, where $ARG_{\mathcal{N}}$ is an ARG for $\mathcal{N}$, $isFull$ is a Boolean flag.
$\phi_I \leftarrow \bigwedge_{v \in V}\{v = \alpha_I(v)\}$
$s_0 \leftarrow (M_I, \phi_I)$
$S \leftarrow \{s_0\}, E \leftarrow \emptyset, N \leftarrow \{s_0\}$
while $N \neq \emptyset$ **do**
  $(M, \phi) \leftarrow Pick(N)$ // Take a node from N
  $N \leftarrow N \setminus \{(M, \phi)\}$
  foreach $t \in T$ *s.t.* $M[t\rangle M'$ **do**
    $\phi' \leftarrow \phi \oplus guard(t)$
    if $\neg(\phi' \sim false)$ then
      foreach *state* (M_o, ϕ_o), *on the path from* s_0 *to* (M, ϕ) **do**
        if $(M' \succ M_o) \wedge (\phi' \sim \phi_o)$ **then**
          return $(\mathbb{S}, E, s_0\mathbb{]} false)$
      $E \leftarrow E \cup \{\mathbb{[}(M, \phi), t, (M', \phi')\mathbb{]}\}$
      if $\forall (\bar{M}, \bar{\phi}) \in S: \bar{M} \neq M' \vee \neg(\phi' \sim \bar{\phi})$ **then**
        $S \leftarrow S \cup \{(M', \phi')\}, N \leftarrow N \cup \{(M', \phi')\}$
return $(\mathbb{S}, E, s_0\mathbb{]}, true)$

Algorithm 1. ConstructARG (N, (MI, αI)).

Since some behavioral properties could be solved as coverability problems, it is makes sense to propose an abstract coverability graph (ACG) for a user. To operate with unbounded nets, we use the special symbol ω, as in [24], which represents an unbounded number of tokens. For each integer n, $\omega > n$, $\omega \pm n = \omega$ and $\omega \geq \omega$. ACG is defined as follows:

Definition 6 (Abstract Coverability Graph). Let $\mathcal{N} = \langle P, T, F, V, guard \rangle$ be a DPN with initial state (M_I, α_I). Let $\Phi(V)$ be the language of constraints, as in Section 3. Abstract Coverability Graph $ACG_{\mathcal{N}}$ of $\mathcal{N}$ is a tuple $\langle S, E, s_0 \rangle$, where:

- $s_0 = (M_I, \phi_I) \in S$ is the initial node with $\phi_I = \bigwedge_{v \in V}\{v = \alpha_I(v)\}$;
- $S \subseteq \mathcal{M}_N \times \Phi(V)$ is the least set that contains s_0 and is closed under the transition relation;
- $E \subseteq S \times T \times S$ is a set of arcs labeled with transitions, s.t. $((M, \phi), t, (M', \phi')) \in E$ if and only if (i) $\phi' = \phi \oplus guard(t)$ and $[[\phi]] \neq \emptyset$, (ii) for each $p \in P$, $M(p) \geq F(p,t)$, (iii) given $M^*(p) = M(p) - F(p,t) + F(t,p)$, $M'(p) = \omega$ if there exists a node $(M'', \phi'') \in S_{CG}$ along the path from s_0 to (M, ϕ), s.t. $(M'', \phi'') \sqsubset (M^*, \phi')$, and $M^*(p) > M''(p)$, otherwise $M'(p) = M^*(p)$.

In some cases, it might be more convenient to investigate coverability problems on a tree structure. For these purposes, we also propose an abstract coverability tree (ACT) for a user, whose definition is very similar to Definition 6, and the only difference is in the graph structure. The algorithms for constructing an ACG and an ACT are similar to Algorithm 1 but guarantee to return the full graph, where unbounded positions are denoted by ω. We skip definitions of these algorithms for brevity.

The structures shown above properly represent all the paths acceptable in a process. For instance, if there is a transition firing $(M, \alpha)[t\rangle(M', \alpha')$, an ARG always contains an arc $((M, \phi), t, (M', \phi'))$ with $\alpha \in [[\phi]]$ and $\alpha' \in [[\phi']]$. However, they may not capture the situations when a transition *may not* fire due to the current variable valuation. The graphs at their current state could be used for verifying model boundedness, absence of dead transitions, presence of paths leading to a final state, presence of deadlocks and livelocks at the backbone level, presence of markings at which neither of the transitions may fire due to input conditions of the transitions, etc.

4.2 Refined State Space Abstractions

When we need a more granular structure, where all the deadlocks induced by the addition of a data flow are shown, we can transform the source net and construct the state space structures from the previous section. The transformation is the following. For each transition t with a non-trivial input condition, we add a complementary transition (denoted as $\tau(t)$) that does not change the marking but has a negated input condition of t as a constraint, i.e., $\neg(\exists write(t)\colon guard(t))$. Fig. 2 illustrates

this transformation for a DPN from Fig. 1. An ARG constructed for such a DPN represents all the model deadlocks as nodes that cannot be leaved by non-tau-transitions, which is shown in Fig. 3.

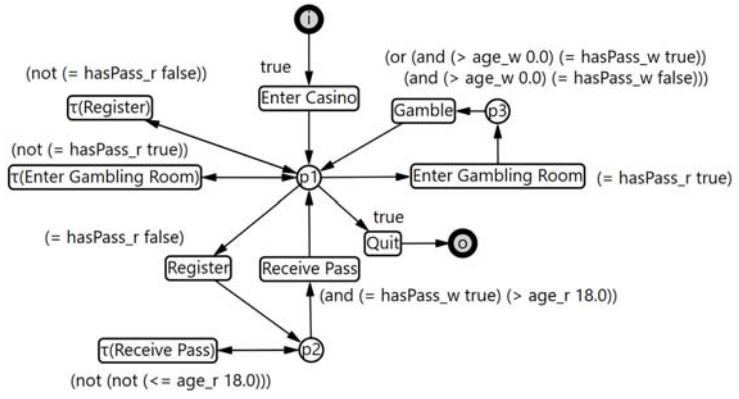

Fig. 2. A DPN from Fig. 1 with added τ-transitions.

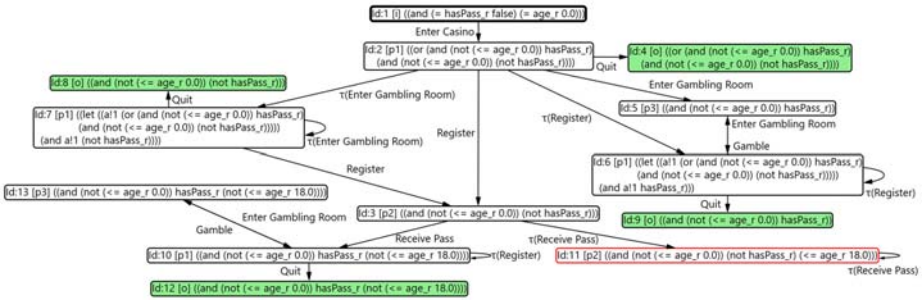

Fig. 3. An ARG for the DPN from Fig. 2. A node with a red border denotes a deadlock. Green nodes denote final states.

However, such an ARG may not capture the livelocks induced by the data flow. The reason is simple: the state-space narrowing caused by firing τ-transitions may be overwritten by a transition in a cycle that updates variables. We need to conduct an extended model transformation to make an ARG granular enough for depicting livelocks. This can be done by splitting transitions that occur in cycles.

In [9], this procedure is called DPN refinement. The proposed approach requires multiple ARG constructions to modify a DPN. At each iteration, for each transition, we need to define a set of transitions based on which the split is done. For this, the proposed approach traverses all the elementary and compound cycles in the ARG. All this together makes the refinement method barely applicable to large models for which an ARG may be of a sufficiently big size.

We introduce a new version of a DPN refinement, shown in Algorithm 2, that requires a single ARG construction and that does not require finding all the elementary cycles in an ARG. The algorithm constructs maximal mutually disjoint cycles exploiting information about reachability sets of each ARG node (compared to finding all the elementary cycles), and splits transitions occurring in cycles. A transition is split based on the input conditions of other transitions occurring in the same cycles or allowing leaving them. The input condition is considered if it includes variables overwritten by the current transition. In the algorithm, we use functions R and R_i to denote mappings between transitions and their refined versions (R for overall and R_i for the current loop iteration). Correspondingly, we use $R^{-1}(t)$ to denote the base version of transition t. For each transition t, we

also define $N(t)$ as a set of transitions that occur in cycles that include t (in $ARG_{\mathcal{N}}$) or allow leaving them. In a while loop, the algorithm splits transitions in cycles until the set of transitions stabilizes. For each DPN transition t, we define the set $T_{dividers}$ as a subset of transitions, whose base versions are in the same cycles as the base version of transition t. Transition t' is in $T_{dividers}$ if t' has an input condition on a variable that is overwritten by t. Procedure $RefineTransition$ (Algorithm 3) splits a transition based on $T_{dividers}$. We try to perform the refinement for each DPN transition and proceed to the next iteration if at least one of the refinements produced multiple transitions. After the set of transitions stabilizes, the loop is exited and the flow relation of the refined DPN is defined. Refined transitions have the same incoming and outgoing arcs as their base versions. Compared to [9], we may split slightly more transitions, but the growth in the resulting ARG size is compensated for by skipping multiple ARG constructions.

Input: A DPN $\mathcal{N} = \langle P, T, F, V, guard \rangle$ with an abstract reachability graph $ARG_{\mathcal{N}}$.
Result: $\mathcal{N}_R = \langle P, T_R, F_R, V, guard \rangle$, a refined version of $\mathcal{N}$.
$toProceed \leftarrow true$, $T_R \leftarrow T$
$Cycles, Exits = GetMaxDisjointCycles(ARG_{\mathcal{N}})$
foreach $t \in T$ **do**
  $R(t) \leftarrow t$
  $N(t) \leftarrow \{t' \in T \mid (s, t', s') \in C \cup Exits(C), (s'', t, s''') \in C, C \in Cycles, t' \neq t\}$
while $toProceed$ **do**
  $toProceed \leftarrow false$
  foreach $t \in T_R$ **do**
    $T_{dividers} \leftarrow \{t' | t' \in R(t''), t'' \in N(R^{-1}(t)), read(t') \cap write(t) \neq \emptyset\}$
    $R_i(t) \leftarrow RefineTransition(t, T_{dividers})$
    if $|R_i(t)| > 1$ **then**
      $toProceed \leftarrow true$
  foreach $t \in T$ **do**
    $R(t) \leftarrow \{t' \mid t' \in R_i(t''), t'' \in R(t)\}$
  $T_R \leftarrow \{t \mid t \in R(t'), t' \in T\}$
foreach $t \in T_R$ **do**
  foreach $p \in P$ **do**
    $F_R(p, t) \leftarrow F(p, R^{-1}(t))$, $F_R(t, p) \leftarrow F(R^{-1}(t), p)$
return $\langle P, T_R, F_R, V, guard \rangle$

Algorithm 2. RefineDPN (N, ARG$_N$).

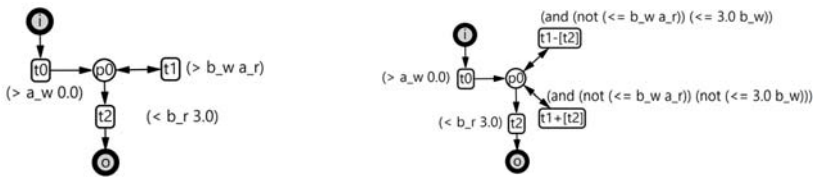

Fig. 4. DPN refinement example. (a) DPN before the refinement. (b) DPN after the refinement. Transition $t1$ is split into $t1 - [t2]$ and $t1 + [t2]$ based on the condition of $t2$.

Algorithm 3 defines the procedure that splits a transition based on the set of transitions-dividers. For each transition t_d in the set of transitions-dividers, we obtain its input condition, substitute the read variables for write ones in this condition if such variables are overwritten by t, and try to split each transition from $T_{refined}$, which accumulates the refined transitions, based on this condition and its negation. The example of the DPN refinement is shown in Fig. 5. Here, transition $t1$ is split based on $t2$. The constraint of $t1$ is $b^w > a^r$ and the only transition from $T_{dividers}$ has the guard $b^r < 3$. The refinement produces two transitions: one with guard $b^w > a^r \wedge b^w < 3$ and another with guard $b^w > a^r \wedge b^w \geq 3$.

If τ-transitions are added to the refined DPN, the ARG for the resulting model becomes sufficient for highlighting all model livelocks and, thus, could be used for soundness verification.

Input: A transition t to refine and a set of transitions $T_{dividers}$ to conduct the refinement based on.
Result: $T_{refined}$, a set of transitions resulted from refining t.
$T_{refined} \leftarrow \{t\}$
foreach $t_d \in T_{dividers}$ **do**
  $\phi_d \leftarrow \exists write(t_d): guard(t_d)$
  foreach $v \in write(t)$ **do**
    $\phi_d \leftarrow \phi_d[v^r \backslash v^w]$
  $T_{refined_d} \leftarrow \emptyset$
  foreach $t_r \in T_{refined}$ **do**
    $guard(t_r^+) \leftarrow guard(t_r) \wedge \phi_d$
    $guard(t_r^-) \leftarrow guard(t_r) \wedge \neg\phi_d$
    if $IsSatisfiable(guard(t_r^+)) \wedge IsSatisfiable(guard(t_r^-))$ **then**
      $T_{refined_d} \leftarrow T_{refined_d} \cup \{t_r^+, t_r^-\}$
    else
      $T_{refined_d} \leftarrow T_{refined_d} \cup \{t\}$
  $T_{refined} \leftarrow T_{refined_d}$
return $T_{refined}$

Algorithm 3. RefineTransition (t, $T_{dividers}$).

4.3 Soundness Verification

The implemented soundness verification algorithm shown in Algorithm 4 transforms the model by splitting transitions and adding τ-transitions and investigates the ARG of the resulting model. The analysis performs graph-traversing techniques to verify the soundness property from Definition 3.

Input: A DPN $\mathcal{N} = \langle P, T, F, V, guard \rangle$ with initial state (M_I, α_I) and final marking M_F.
Result: $(ARG, isSound)$, where ARG is an ARG for $\mathcal{N}$ (if unbounded) or for $\mathcal{N}_{R_\tau}$ (otherwise), $isSound$ is a Boolean flag.
$(ARG_{\mathcal{N}}, isFull) \leftarrow ConstructARG(\mathcal{N}, (M_I, \alpha_I))$
if $\neg isFull$ **then**
  return $(ARG_{\mathcal{N}}, false)$
$\mathcal{N}_R \leftarrow RefineDPN(\mathcal{N}, ARG_{\mathcal{N}})$
$\mathcal{N}_{R_\tau} \leftarrow GetTauDPN(\mathcal{N}_R)$
$ARG_{\mathcal{N}_{R_\tau}} \leftarrow ConstructARG(\mathcal{N}_{R_\tau}, (M_I, \alpha_I))$
return $AnalyzeARG(ARG_{\mathcal{N}_{R_\tau}}, M_F)$

Algorithm 4. VerifySoundness (N, (M_I, α_I), M_F).

The examples of soundness verification results visualization are shown in Fig. 3 and Fig. 7.

Besides classical soundness verification, the tool implements an algorithm for verifying relaxed-lazy soundness, which is applicable for checking correctness of resource-oriented models [13]. Relaxed lazy soundness requires a model to have at least some executions that terminate with one token in o and potentially other tokens in the model and have each of the transitions present in at least one such execution. For verifying this property, classical ACG becomes sufficient.

4.4 ASML Format

Although ARG visualization may be convenient, it may be useful to export the graph for further analysis. This is especially reasonable if the number of ARG nodes is too high for visualizing or if it is undesirable to reconstruct the state space again. That is why we have proposed the ASML format for exporting and importing the graphs of abstract state space abstractions. The full xsd-schema for this format is stored in the GitHub repository. In short, we store the information about nodes and arcs, about the state space type (ACG, ARG, ACT, etc.), and about the DPN elements, namely transitions, variables, and the final marking. Transition constraints follow the Backus-Naur form from Section 3.1. State constraints have a very similar form with the only difference that variables in formulas do not have `_w` and `_r` subscripts. The ASML format is sufficient both for analyzing graphs in other systems and for their proper state space visualization in the DPN Verifier.

5. Means for Data Petri Net Repair

If a model is unsound, it is often desired to slightly modify the model to make it sound. In our tool, we follow the approach proposed in [10], repairing a model by restricting transition constraints (by that, we forbid executions that lead to failure points in the source model). A repair can be divided into three steps: (i) making the DPN bounded, (ii) eliminating model deadlocks and livelocks, (iii) removing dead transitions and isolated places. The algorithm is implemented as a semi-decision procedure: it guarantees to terminate, but the repair may not succeed. If the repair was not successful, the tool informs the user about it and proposes to use other repair algorithms. We omit the algorithm pseudocode for brevity since it is very similar to the code given in [10]. The difference in the algorithm implementation is the following. First, our implementation uses an ACG to make the DPN bounded instead of an ACT. A coverability tree is usually larger than a coverability graph and has duplicate nodes. By using an ACG, we may omit the necessity to compute the results of the same expressions multiple times. Second, our implementation incorporates the DPN refinement from Algorithm 2, which requires a single ARG construction to refine a model and does not require finding all the elementary cycles. This allows conducting the repair faster in most cases.

Fig. 5 illustrates the repair result of a DPN from Fig. 1.

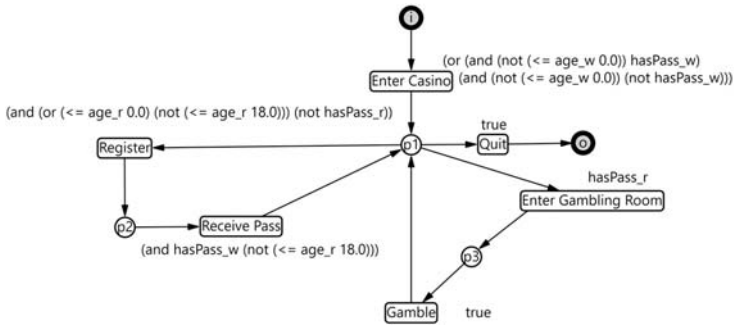

Fig. 5. A repaired DPN from Fig. 1 using the implemented repair algorithm.

6. Delivery Formats

The DPN Verifier toolkit is delivered as a library, as a console application, and as a desktop application. Compared to Ada [12], DPN Verifier is fully open-source and easily deployable, which guarantees its usefulness even after the maintenance termination. The toolkit is implemented using the C# language, allowing for generally faster algorithm execution compared to Python.

6.1 Library

The solution consists of four main libraries: *DPN.Models, DPN.Generation, DPN.Parsers,* and *DPN.Soundness. DPN.Models* is a core library containing classes for all the elements associated with DPNs used by other libraries. *DPN.Generation* is a library for generating random DPNs for experiments. It provides capabilities for constructing a random DPN according to the given parameters. The library could be used for synthetic experiments of algorithms for DPNs or as an entry-point to see how a DPN can be verified, repaired, and exported. *DPN.Parsers* is a library that allows converting a DPN to the PNMLX format, an abstract state space structure to the ASML format, and vice versa. The library also includes the internal code for proper serialization and deserialization of formulas, both in transition and state constraints. *DPN.Soundness* is a library that provides capabilities for soundness verification and repair. Verification algorithms (classical and relaxed-lazy) implement a single method, *Verify*, that takes a DPN and verification settings as input and returns the object that stores all the information regarding the DPN soundness that can be further

shown to a user. A repair algorithm is implemented only for classical soundness. Its method *Repair* takes a DPN and repair settings as an input and returns an object storing the important information regarding the repair algorithm execution. Verification and repair settings are used to configure the algorithms, for instance, by specifying the type of a state space structure to use in the operation. Besides that, there are utility classes for constructing state space structures from Section 4.1 and performing transformations from Section 4.2.

6.2 Console Application

The console application, called *DPNVerifier.Console*, provides capabilities for verifying and repairing DPNs saved in the PNMLX format. If verification is called, the tool, as a result, returns information about model soundness and about the abstract state space constructed. If the flag *SaveStateSpace* is used, the state space is saved in the ASML format at the predefined location. If the flag *Verbose* is used, the information about each node in the abstract state space structure is written as an output. If a repair is called, the tool, as a result, produces the repaired DPN (if a repair was successful) and saves it in the PNMLX format at the predefined location. The console app could be used as a standalone application or in different pipelines. Regarding pipelines, we define two main directions: (i) using the verify and repair operations right after automatic model discovery to guarantee model soundness, or (ii) using the repair operation together with other DPN repair approaches to propose different repair options to a user for an unsound DPN.

Listing 3 is the output of the *DPNVerifier.Console* call without any arguments, showing all possible parameters for a tool and examples of the tool calls. Listing 4 is the repair result for the DPN example from Fig. 1.

6.3 Desktop Application

The desktop application allows a user to import/export/generate a DPN, visualize it, conduct DPN transformations described in Section 4.2, construct and visualize abstract state space structures, defined in Section 4.1, import/export them, and perform soundness verification and repair.

The main application window is shown in Fig. 6. The *File* tab allows opening and saving a DPN and opening an abstract state space structure. The *Transition Systems* tab allows constructing and visualizing an ARG, an ACG, and an ACT, both for source DPNs and the ones transformed using techniques from Section 4.2. The *Soundness* tab provides capabilities for soundness verification and repair. The *Model* tab provides capabilities for transforming a DPN to tau-DPN and refined DPN.

```
DPN Verification and Repair Console Application

Usage:
  --Operation <Verify|Repair>
  --DpnFile <path_to_dpn_file>
  --OutputDirectory <output_directory>
  --SoundnessType <Classical|RelaxedLazy>
  --SaveStateSpace <true|false>
  --VerificationParameters "<key1> <value1> <key2> <value2>..."
  --RepairParameters "<key1> <value1> <key2> <value2>..."
  --Verbose

Examples:
    --Operation Verify --DpnFile model.pnmlx –OutputDirectory
              ./results --SoundnessType RelaxedLazy --Verbose
    --Operation Repair --DpnFile model.pnmlx –OutputDirectory
              ./results --SoundnessType Classical
```

Listing 3. Admissible expressions definition.


```
Processing DPN: Casino.pnmlx
Places: 5, Transitions: 6, Variables: 2
Starting repair...
Repair result: Success
Repair steps: 1
Repair time: 0.13 seconds
Repaired DPN saved to C:\workspace\Casino-repaired.pnmlx
```

Listing 4. Admissible expressions definition.

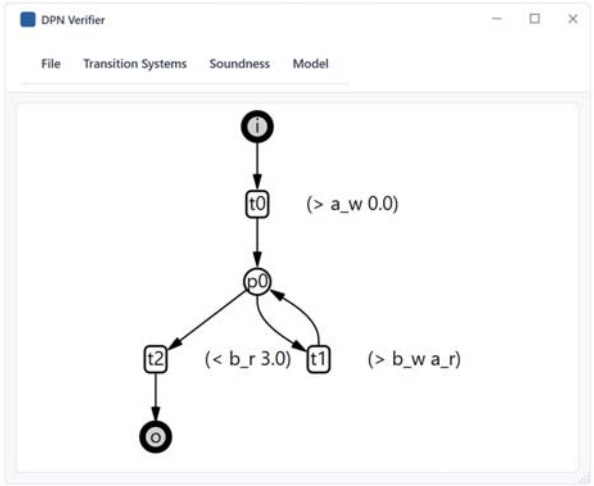

Fig. 6. Main DPN Verifier window with the visualized livelock example from Fig. 4(a).

The visualization of abstract state space structures is done in a separate window, which is also used for illustrating soundness verification results. Fig. 7 shows this window as a result of soundness verification for a DPN from Fig. 6. All the nodes have adjusted visualization according to their type. For instance, final nodes are colored in green, nodes with no path to finals are not colored but have a red border, dead nodes are colored in red, and nodes with unclean finals are colored in blue. At the bottom, the total information regarding soundness is shown. Since the analysis of the state space structure costs sufficiently less than its construction, the analysis is conducted on each state space construction. This helps a user detect potential problems in a model. This window allows a user to export the state space structure in the ASML format by pressing the button *Export Graph*. The exported graph can further be opened from the main window. If the graph is too large, we only visualize it if a user approves this action.

7. Experimental Evaluation

We tested our verification and repair algorithm implementations in the DPN Verifier toolkit on the standard DPNs from the literature and compared their performance against two existing tools ([10] and [12]). As shown in Table 1, our implementation is faster for almost all models. The sole exception was the Road Fines Mined model, for which our repair algorithm generated significantly more transition splits than the method in [10], resulting in a slightly higher execution time. Despite this case, the experimental results demonstrate that our refinement approach generally reduces verification and repair times for standard models from the literature and that the implemented soundness verification and repair algorithms can be used in realistic application domains.

In the future, we also plan to evaluate the algorithms' implementations on synthetic data using the DPN generation feature described in the previous section.

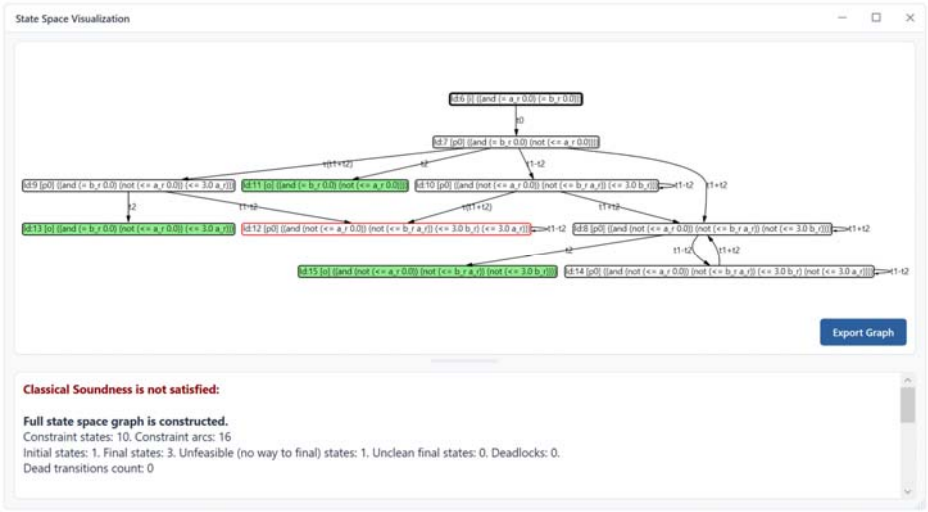

Fig. 7. State Space Visualization window with results for soundness verification of the livelock example.

Table 1. Verification and repair times on models from the literature. Size is measured in the number of places (P), transitions (T), and variables (V). The unbounded and BPMN examples were not tested on the Ada tool. Nonetheless, the unbounded example cannot be repaired by the Ada tool due to the algorithm restriction.

Model	Size	Verification			Repair		
		DPN Verifier	Prototype [10]	Ada [12]	DPN Verifier	Prototype [10]	Ada [12]
Livelock Example [9]	3P+3T+2V	77 ms	196 ms	855 ms	123 ms	205 ms	2.1 s
Unbounded Example [10]	5P+4T+1V	5 ms	7 ms	?	20 ms	32 ms	-
BPMN Example [9]	10P+10T+5V	38 ms	54 ms	?	116 ms	203 ms	?
Casino Example [10]	5P+6T+2V	31 ms	40 ms	920 ms	112 ms	134 ms	2.7 s
Digital Whiteboard: Transfer [25]	7P+6T+3V	20 ms	26 ms	120 ms	68 ms	118 ms	2.1 s
Package Handling [8]	16P+28T+5V	745 ms	902 ms	1.3 s	754 ms	3.7 s	6 s
Road Fines Mined [25]	9P+19T+8V	296 ms	410 ms	3.1 s	1.8 s	1.6 s	24 s
Simple Auction [16]	4P+4T+2V	57 ms	183 ms	1.7 s	130 ms	263 ms	2.5 s
Hospital Billing [25]	17P+16T+4V	68 s	108 s	181 s	Already Sound		
Sepsis Mined [25]	24P+35T+4V	45 s	101 s	103 s	Already Sound		

8. Conclusion

In this paper, we introduced the toolkit for soundness verification and repair for data-aware process models represented as DPNs. The toolkit allows importing a DPN, visualizing it, verifying it for soundness, and repairing if it is unsound. Besides that, the state space of the DPN can be visualized at different granularity levels. The latter allows for verification of a desired behavioral property without the need to construct the most granular state space abstraction. The state space structure itself could be exported, which can be especially useful if a user wants to analyze the state space for

different properties or if a user wants to save the results in order not to construct it again. The verification and repair algorithms introduced in this paper and implemented in the tool are justified to have lower execution times compared to the existing solutions. The toolkit is delivered in three forms: as a library, as a console application, and as a desktop application.

The toolkit could be used in different scenarios. First, when a modeler designs a process to be implemented. Verifying its model for soundness could help to detect failure points in a process before its implementation, whereas repairing the model could eliminate such failure points and guarantee that the process would always properly terminate. Second, when a process is already implemented according to the model, and an analyst or a modeler wants to verify the already used process for the absence of failure points or of potentially unbounded resource consumption. The toolkit can highlight the problems in a model if they exist and propose a solution for eliminating such problems. This information can form a basis for decision-making regarding process improvement. Third, when a model is automatically discovered from event logs. In this case, soundness of the discovered model is not guaranteed, but can be ensured by verifying and repairing it with the proposed toolkit. It is important to note that the toolkit can also be used for analyzing other behavioral properties, not limited to soundness. The tool can construct the needed state space abstraction, which a user may export and verify for certain properties using other tools.

In the future, we plan to extend the tool with new means for model analysis, potentially using temporal logics. It might also be useful to develop the DPN editor so that a user can build the DPN right in the introduced desktop application. Another important point is to conduct a thorough comparison with [9] and [10] on synthetic data to clarify whether the proposed algorithm adjustments indeed help to reduce the execution time in the vast majority of cases.

References

[1]. Sidorova N., Stahl C., Trcka N. Soundness verification for conceptual workflow nets with data: Early detection of errors with the most precision possible. Information Systems, 36(7), 2011, pp. 1026-1043.
[2]. Tao X., Liu G., Yang B., Yan C., Jiang C., Workflow nets with tables and their soundness. IEEE Transactions on Industrial Informatics, 16(3), 2020, pp. 1503-1515.
[3]. de Leoni M., van der Aalst W. M. P. Data-aware process mining: Discovering decisions in processes using alignments. In Proceedings of the 28th Annual ACM Symposium on Applied Computing, 2013, pp. 1454-1461.
[4]. Aagesen G., Krogstie J. Bpmn 2.0 for modeling business processes. Handbook on Business Process Management 1: Introduction, Methods, and Information Systems. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 219-250.
[5]. van der Aalst W. M. P. Verification of workflow nets. In Application and Theory of Petri Nets 1997, 1997, pp. 407-426.
[6]. de Leoni M., Felli P., Montali M. A holistic approach for soundness verification of decision-aware process models. In Conceptual Modeling, 2018, pp. 219-235.
[7]. Felli P., de Leoni M., Montali M. Soundness verification of data-aware process models with variable-to-variable conditions. Fundamenta Informaticae, 182(1), 2021, pp. 1-29.
[8]. Felli P., Montali M., Winkler S. Soundness of data-aware processes with arithmetic conditions. In Advanced Information Systems Engineering, 2022, pp. 389-406.
[9]. Suvorov N. M., Lomazova I. A. Verification of data-aware process models: Checking soundness of data petri nets. Journal of Logical and Algebraic Methods in Programming, vol. 138, 2024, 100953.
[10]. Suvorov N. M., Lomazova I. A. Soundness Correction of Data Petri Nets. IEEE Access, vol. 13, 2025, pp. 149142-149157.
[11]. Zavatteri M., Bresolin D., de Leoni M. Repair of unsound data-aware process models. Business Process Management Workshops. Cham: Springer Nature Switzerland, 2024, pp. 383-395.
[12]. Felli P., Montali M., Winkler S. Repairing soundness properties in data-aware processes. In 2023 5th International Conference on Process Mining (ICPM), 2023, pp. 41-48.
[13]. Suvorov N. M., Lomazova I. A. Relaxed Lazy Soundness Verification for Data Petri nets. Proceedings of the Institute for System Programming of the RAS, 37(4), 2025, pp. 69-84.

[14]. Mendling J., Neumann G., van der Aalst W. Understanding the occurrence of errors in process models based on metrics. In Proceedings of the 2007 OTM Confederated International Conference on the Move to Meaningful Internet Systems: CoopIS, DOA, ODBASE, GADA, and IS Volume Part I, ser. OTM'07. Berlin, Heidelberg: Springer-Verlag, 2007, p. 113-130.
[15]. Roy S., Sajeev A., Bihary S., Ranjan A. An empirical study of error patterns in industrial business process models. IEEE Transactions on Services Computing, 7(2), 2014, pp. 140-153.
[16]. Roy S., Sajeev A., Gopichand A., Bhattacharya A. An empirical analysis of diagnosis of industrial business processes at sub-process levels. In 2016 IEEE International Conference on Services Computing (SCC), 2016, pp. 195-202.
[17]. Mendling J., Verbeek H., van Dongen B., van der Aalst W. M. P., Neumann G. Detection and prediction of errors in EPCs of the sap reference model. Data and Knowledge Engineering, 64(1), 2008, pp. 312-329.
[18]. de Leoni M., Munoz-Gama J., Carmona J., van der Aalst W. M. P. Decomposing alignment-based conformance checking of data-aware process models. In On the Move to Meaningful Internet Systems: OTM 2014 Conferences. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 3-20.
[19]. Felli P., Gianola A., Montali M., Rivkin A., Winkler S., Data-aware conformance checking with smt. Information Systems, vol. 117, 2023, 102230.
[20]. van der Aalst W. M. P., van Dongen B. F., Günther C. W., et al. Prom 4.0: Comprehensive support for real process analysis. In Petri Nets and Other Models of Concurrency. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 484-494.
[21]. Berti A., van Zelst S., Schuster D. PM4PY: A process mining library for python. Software Impacts, vol. 17, 2023, 100556.
[22]. Billington J., Christensen S., Van Hee K., et al. The petri net markup language: Concepts, technology, and tools. In Proceedings of the 24th International Conference on Applications and Theory of Petri Nets, ser. ICATPN'03, Eindhoven, The Netherlands: Springer-Verlag, 2003, pp. 483-505.
[23]. de Moura L., Bjørner N. Z3: An efficient smt solver. In TACAS. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337-340.
[24]. Murata T. Petri nets: Properties, analysis and applications. Proceedings of the IEEE, 77(4), 1989, pp. 541-580.
[25]. Mannhardt F. Multi-perspective process mining. Ph.D. dissertation, Eindhoven University of Technology, 2018.

Информация об авторах / Information about authors

Николай Михайлович Суворов – аспирант аспирантской школы по компьютерным наукам НИУ ВШЭ, стажер-исследователь лаборатории процессно-ориентированных информационных систем (ПОИС) факультета компьютерных наук НИУ ВШЭ. Сфера научных интересов: теория автоматов, распределенные процессы, верификация и исправление моделей.

Nikolai Mikhailovich SUVOROV – postgraduate student at Doctoral School of Computer Science, HSE University, research fellow at Laboratory of Process-Aware Information Systems, Faculty of Computer Science, HSE University. Research interests: automata theory, distributed processes, and model verification and repair.